University of Maryland - College Park

ENPM613-0101,SW01: Software Design & Implementation Project

Fall 2023

SOFTWARE ARCHITECTURE AND DETAILED DESIGN DOCUMENT (SADD)

TEAM 9

LanguageLift

Elevate Your Language Skills

Anjaneya Ketkar
Divya Lnu
Janesh Hasija
Jayesh Pamnani
Manav Gupta

TABLE OF CONTENTS

PRODUCT OVERVIEW

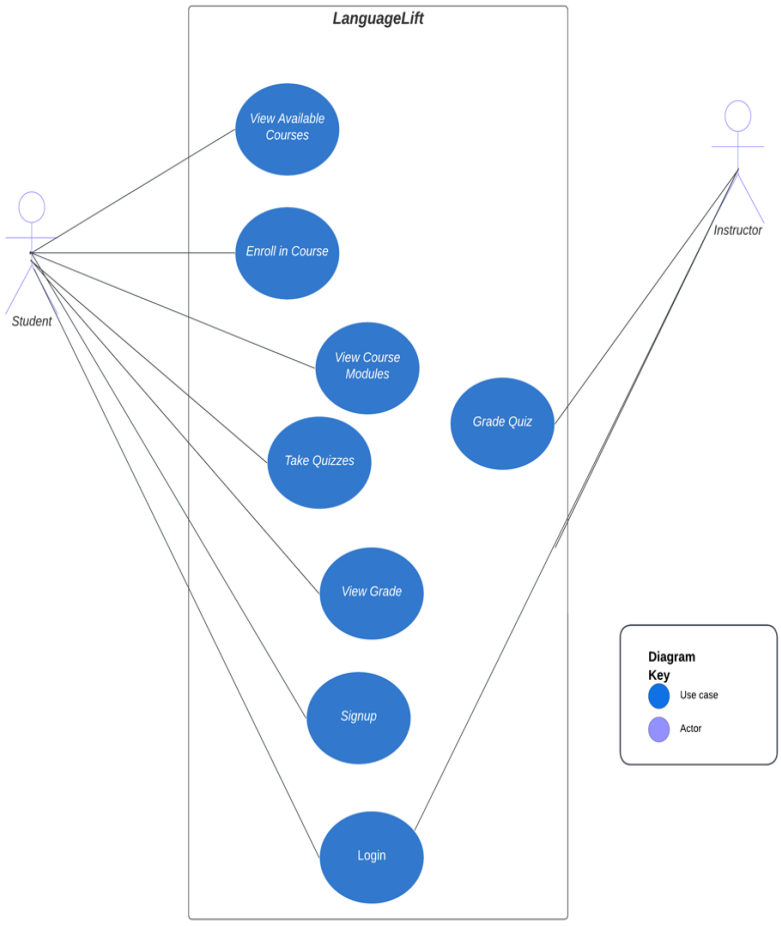
LanguageLift is a language learning software system with the primary function and purpose of enhancing users' language skills. It offers a personalized learning experience tailored to individual goals and proficiency levels. The system provides a wide selection of language courses, incorporating audio, video, quizzes, and real-world scenarios to create immersive lessons. Users can track their progress, participate in a community, and earn achievements. LanguageLift prioritizes accessibility across different devices to accommodate users' busy schedules, ultimately aiming to enrich lives, expand cultural horizons, and foster global connections through improved language skills.

ARCHITECTURE DRIVERS

The architectural decisions have been made keeping in mind various architectural drivers which include: -

- Functional Requirements

  The top priority functionalities that have been taken care of while designing the system are showcased in the given Use Case Diagram



- Quality Attributes

  The top priority quality scenarios that have been taken care of while designing the architecture of the system are as follows: -
  - Performance
  - Usability
  - Availability
  - Security

Quality Attribute	Quality Scenario Name	Quality Scenario Brief Description	Quality Scenario Utility (importance) to the users (Low = 1, Medium = 2, High = 3)	Estimated Quality Scenario development difficulty or risk (Difficult or high risk = 1, Medium difficulty or risk = 2, Rather easy and low risk = 3)	Scenario priority score (utility * difficulty)
Performance	Optimize Course Information Load Time	**Source**: A user **Stimulus:** Tries to search the details of a particular course Artifact: System **Environment:** Normal operation **Response**: The course information page should load **Response Measure:** The course information page should load in under 20 ms	3	3	9
Usability	Easy registration	**Source**: A user **Stimulus**: Tries to signup on the application **Artifact**: System **Environment**: Normal operation **Response**: A user should be able to register on the website **Response Measure:** An average user should be able to register on the website in under 2 minutes under normal conditions	3	3	9
Availability	Software Uptime	**Source:** A user **Stimulus**: Tries to use the application **Artifact:** System Environment: Normal operation **Response:** The user should be able to use the application **Response Measure:** The uptime for the application should be more than 97%	3	2	6

Security	Protection of user credentials	**Source:** An attacker **Stimulus**: Tries to gain unauthorized access to data **Artifact:** System **Response:** The application should store user credentials in encrypted form. **Response Measure:** An attacker must not be able to decrypt the encrypted user credentials 99% of the time.	3	2	6

ARCHITECTURE KEY DECISIONS AND RATIONALE

Below are some of the crucial decisions and the rationale for selecting the architecture:

1. **Client Server Architecture:** Below are the following reasons for the selection of this architecture style:
    a. To centralize data administration, guarantee data consistency, boost security, and enable access across several platforms, a client-server architecture is used. With this design, administrators, instructors, and students may access the Language Lift from a variety of places and devices, meeting their demands.
    b. Teachers, administrators, and students all have distinct responsibilities and rights in LanguageLift. Role-based access control can be implemented using a client-server architecture, guaranteeing that every user group has access to the right capabilities and information while upholding security and privacy.
    c. The number of users, courses, and data will increase as LanguageLift expands. Scalability can easily be achieved via a client-server design by increasing the number of compute instances, which can effectively handle additional users and data. This is essential to satisfy the changing demands of instructors, administrators, and students.
    d. Language Lift must have complete security since it manages important student and teacher data. Secure measures can be centrally managed via a client-server architecture, which enables strong access control, encryption, and authentication. This improves privacy compliance and data protection.
    e. Discussion boards, messaging, and alerts, which are crucial for interaction and engagement between students, professors, and administrators, are supported by the client-server architecture.

2. **Microservices:** For students, instructors, and administrators, creating a Language Lift with a microservices architecture can have several benefits:
    a. **Scalability and Elasticity:** Because microservices are scalable, it is simple to scale individual services to meet changing demands and use patterns.
    b. **Modularization:** LanguageLift's software is divided into distinct subsystems for each of its constituent parts, including User Management Service, Enrollment Management Service, Course Management Service, and Quiz Management Service. Because each microservice can handle a particular facet of functionality, managing, updating, and growing the software is made easier.
    c. **Service Isolation and Fault Tolerance:** Other components of the system continue to function normally in the case of a service outage, ensuring that student and instructor activities continue. For instance, if the quiz management service goes down then the rest of the components like course content management will continue to work.

3. **Model View Controller:** Below are the reasons for the selection of this architecture:
    a. **Separation of Concerns:** MVC enforces a clear separation of concerns within an application**.**
    b. **Testability:** Because of the clear separation of concerns, it's easier to write unit tests for each component, ensuring that they work as expected.
    c. **Maintainability:** The separation of concerns makes the codebase more maintainable.

On the other hand, Language Lift built with a monolithic architecture has several disadvantages and restrictions that may make it less than ideal for such a dynamic and intricate system. For this website application, a monolithic or shared data design might not be the ideal option for the following reasons:

- Scalability: It might be challenging to scale a monolithic Language learning app to handle an increasing user base. Vertical scaling, or updating individual servers, is frequently necessary. This can be expensive and eventually run into technological constraints. Maintaining good performance and availability for an expanding user base will be challenging without a sophisticated and costly infrastructure.
- In a monolithic design, integrating with other services or tools—like content providers or language assessment tools—can be more difficult. Microservices and service-oriented architectures, on the other hand, provide more flexibility when it comes to integrating external services.

In conclusion, while monolithic architecture could be effective for less complex systems, creating Language Lift using one frequently proves to be difficult and inefficient. The website's capacity to adjust to shifting user demands and trends in education may be hampered by its constraints in terms of scalability, flexibility, and maintenance. On the other hand, more recent architectural approaches, such as microservices or service-oriented architectures, provide a more flexible and scalable means of creating a system that can better satisfy the needs of both learners and educational institutions.

In the context of LanguageLift, the selected technologies—MySQL, Angular, and Spring Boot—have been carefully chosen to meet specific requirements and objectives. Insights into the technology choices and considerations along with associated trade-off analyses are given below:

Technology Considerations:

1. MySQL:
    a. Relational Database Management: MySQL was chosen for its strong support for structured data and relational database management. This is essential for storing user profiles, progress tracking data, course content, and other structured information.
    b. Data Integrity: MySQL's ACID compliance ensures data integrity, a crucial factor for a language learning platform where user progress and achievements must be accurately recorded and maintained.

2. Angular:
    a. User Interface Development: Angular is a robust front-end framework chosen for its ability to create dynamic and interactive user interfaces. This is vital for delivering an engaging and intuitive learning environment in LanguageLift.
    b. Responsive Design: Angular supports responsive web design, ensuring the platform's accessibility and usability across various devices and screen sizes.

3. Spring Boot:
    a. Back-End Development: Spring Boot is selected for building the back-end of LanguageLift due to its ease of development, scalability, and robustness.
    b. Microservices Architecture: Spring Boot facilitates the implementation of a microservices architecture, enabling scalability, fault tolerance, and modular development.

Reuse Considerations:

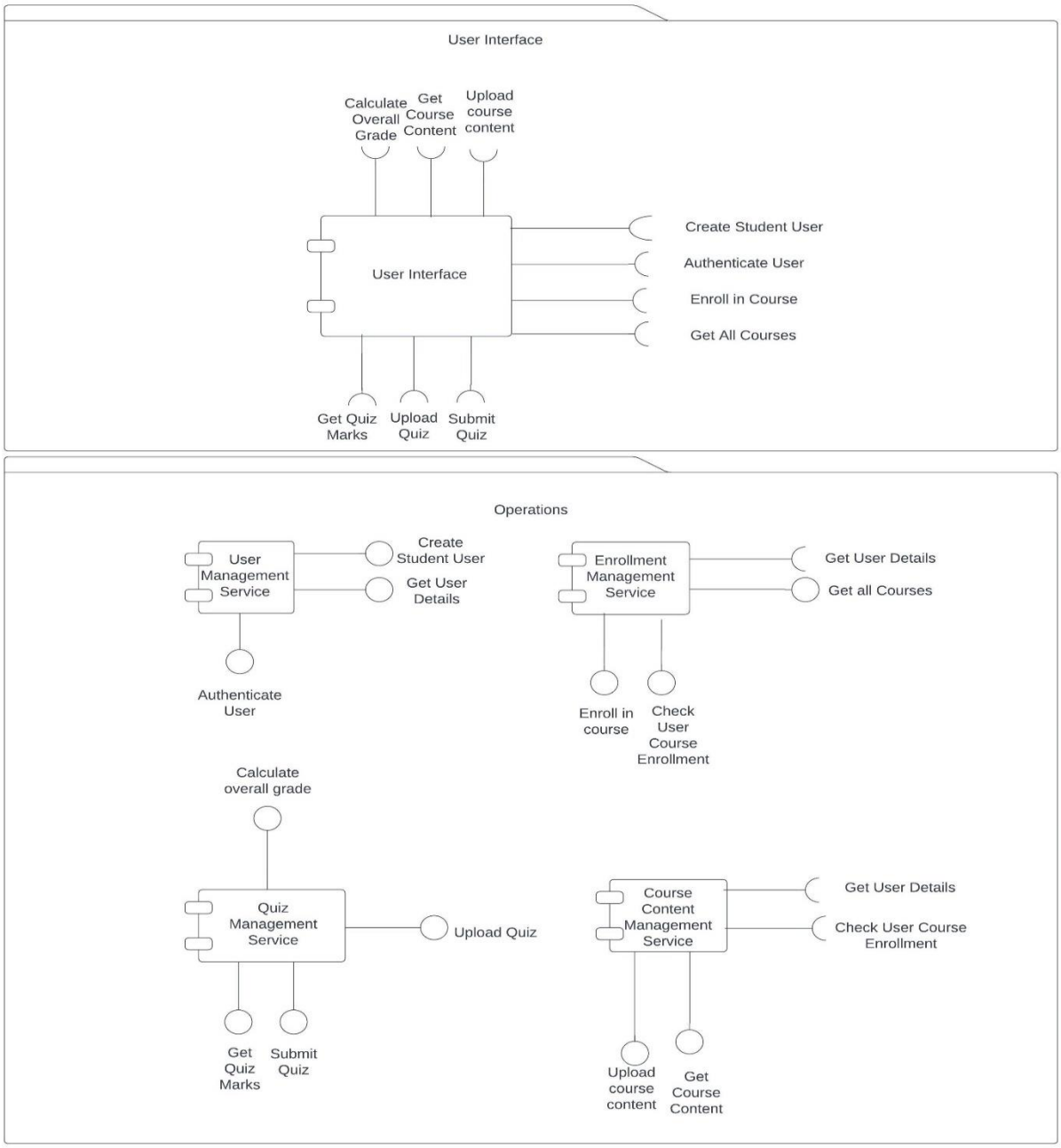
1. The LanguageLift development team will explore opportunities for code reuse within the application, especially when building and maintaining various language course modules. This reuse will help maintain consistency and reduce development effort.
2. Reuse of user authentication and authorization modules can enhance security and user management across different components of the system.
3. For scalability and performance optimization, components of the software can be designed with the potential for reuse and adaptation as the user base grows.

Trade-off Analyses:

As of the present, the technology landscape relevant to this project encompasses prominent choices in the database, server framework, and front-end framework domains. For relational databases, MySQL holds a dominant position, while in the realm of non-relational databases, MongoDB (NoSQL) is widely adopted. Among server frameworks, Java Spring Boot and Python Flask emerge as popular options, catering to diverse programming preferences. When it comes to front-end development, Angular and React stand out as widely embraced frameworks, offering a multitude of tools and libraries for building responsive and interactive user interfaces.

	Factors → (Weight) Technologies ↓	Integration with each other (25%)	Technology team's familiarity with the software(50%)	Ease of implementability (25%)	Total Score
Backend	Java Spring Boot	90	100	70	90
	Python Flask	90	20	100	57.5
Database	MySQL	90	100	70	90
	NoSQL	90	30	90	60
Frontend	Angular	90	70	80	77.5
	React	90	20	70	50

ARCHITECTURE LOGICAL VIEW



Component 1
Component: User Management Service
Component Responsibilities: Performs all operations involving creating accounts, authenticating, and fetching details from the user table.
Provided interfaces: Get User Details, Create Student Account, Authenticate users.
Required Interface: None.

Interface Description:
Interface Name: Authenticate users.
Interface responsibilities: This interface compares the username and password entered by the user and compares it with the one present in the database.
Interface Operations: Query database, Decrypt the queried data, and compare the data.

Operation Description:

Query Database	
Operation signature	Username: String, Password: String
Operation description	Queries the database
Operation preconditions	Username and password are not null, and the user is registered
Operation postconditions	Username and password are present in the database, or the user is not authenticated.

Component 2
Component: Enrollment Management Service
Component Responsibilities: Performs all operations involving enrollment of students in any course and updating the course table.
Provided interfaces: Get all courses, Enroll in Courses, and Check user Course Management.
Required Interface: Get User Details.

Interface Description:
Interface Name: Get all courses.
Interface responsibilities: This interface returns all the available courses.
Interface Operations: Query database.

Operation Description:

Query Database	
Operation signature	
Operation description	Queries the database
Operation preconditions	At least 1 course should be present in the courses table of database
Operation postconditions	Returns all the courses in course table

Component 3
Component: Quiz Management Service
Component Responsibilities: Performs all operations involving the quizzes including uploading the quiz, grading, and updating quiz submission.
Provided interfaces: Upload Quiz, Submit Quiz, Get Quiz Marks Score, calculate overall grade
Required Interface: None

Interface Description:
Interface Name: Upload Quiz
Interface responsibilities: This interface allows the instructor to upload a quiz into the LMS module for a particular course.
Interface Operations: Create Record in the database.

Operation Description:

Create Record in the database.	
Operation signature	QuizData: String, InstructorID: Integer, CourseID: Integer
Operation description	Creates a new record in the database.
Operation preconditions	The instructor has proper access to upload data into that course.
Operation postconditions	The Quiz is uploaded into the database and visible to the students.

Component 4
Component: Course Content Management Service
Component Responsibilities: Performs all operations involving uploading, updating, and deleting the course content and modules.
Provided interfaces: Upload course content, Get Course Content.
Required Interface: Get User Details and check User Course Enrollment.

Interface Description:
Interface Name: Upload Course Content.
Interface responsibilities: This interface is responsible for facilitating content uploads and organization.
Interface Operations: Adding and Updating course-related materials.

Operation Description:

Adding course-related materials	
Operation signature	InstructorId: Integer, CourseId: Integer, Content: String
Operation description	Adds/Updates existing course content with user-provided metadata
Operation preconditions	Checks if specific course content is available within a course.
Operation postconditions	Adds new material to the database

Component 5
Component: User Interface
Component Responsibilities: Provides responsive interaction for the user. Takes commands from the user, passes them to the servers, and displays the response from these servers.
Provided interfaces: None.
Required Interface: Get Course Content, Upload course content, Create Student User, Authenticate User, Enroll in Course, get all Courses, Submit Quiz, Upload quiz, Get quiz marks, calculate overall grade.

Interface Description:
Interface Name: Enroll in Course
Interface responsibilities: This interface needs to be simple for the users to be able to easily enroll in the courses of their choice.
Interface Operations: check user course enrollment, and send new enrollment data to the Enrollment Management service

Operation Description:

Send new Enrollment Data to the Enrollment Management Service	
Operation signature	User ID: Integer, Course ID: Integer
Operation description	API calls to Enrollment management service
Operation preconditions	User ID and Course ID are not null, and the user is registered
Operation postconditions	Status 200 returned from Enrollment Management Service

In our software architecture, we've implemented a layered approach with distinct components to address various quality requirements.
Here's a breakdown of how our components fit into the selected styles:

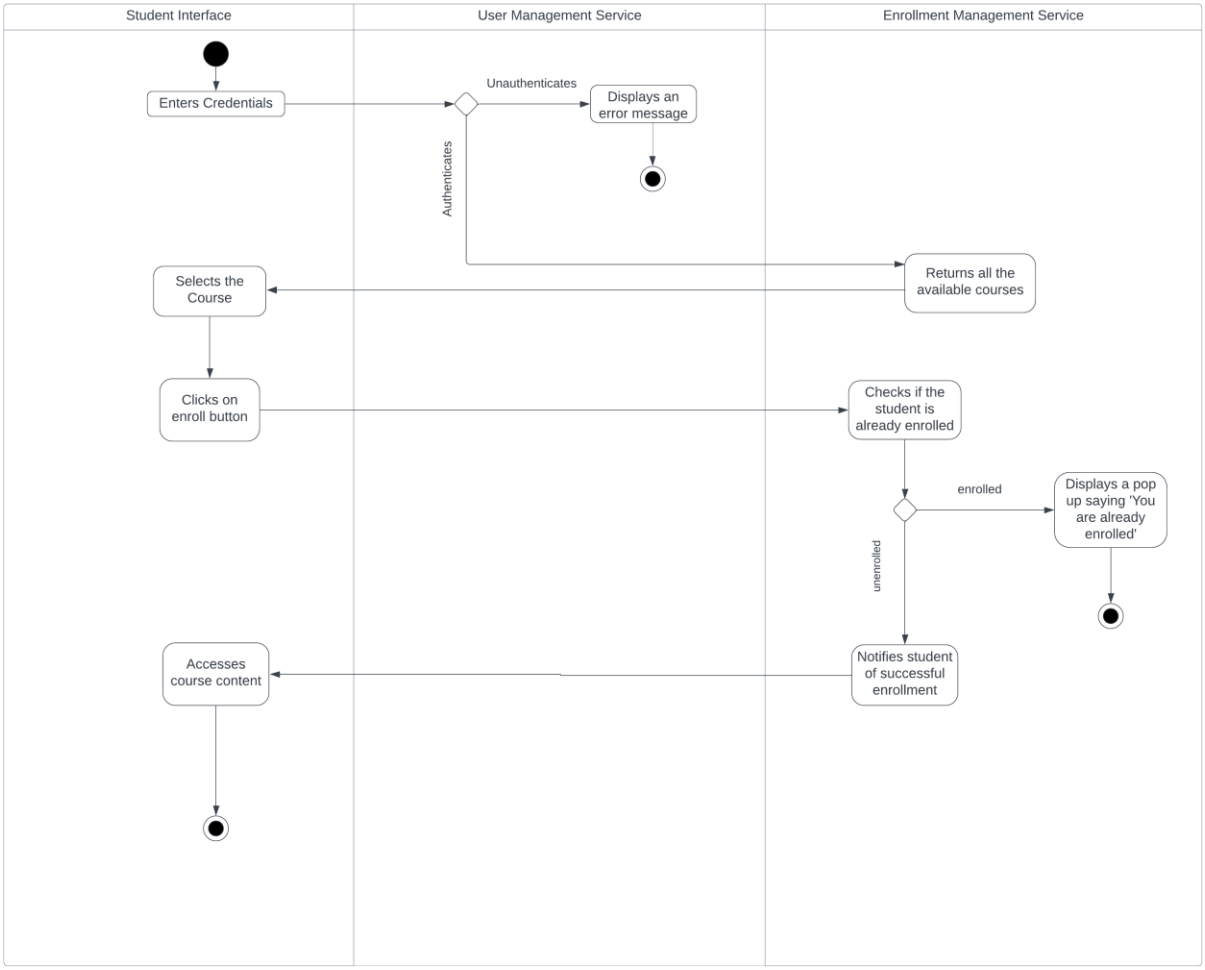
- Server Layer:
    - Microservices: The server layer comprises four microservices - User Management Service, Grade Management Service, Enrollment Management Service, and Course Content Management Service. Each microservice is designed to handle specific tasks and responsibilities. This follows the microservices architectural style, which promotes modularity and separation of concerns.
- Client:
    - The frontend serves as the client and uses controllers from the backend server to manage user interactions. This follows the client-server architectural style, where the frontend acts as a client connected to the backend server.
- User Interface (View):
    - The user interface is where end-users interact with our software. It represents the presentation layer, following the MVC (Model-View-Controller) architectural pattern.
- Backend Server (Controller & Model):
    - The backend server is responsible for controlling the business logic (controller) and managing the data (model). This adheres to the MVC architectural pattern, ensuring separation of concerns and maintainability.

Here's a breakdown of the tactics we have identified:

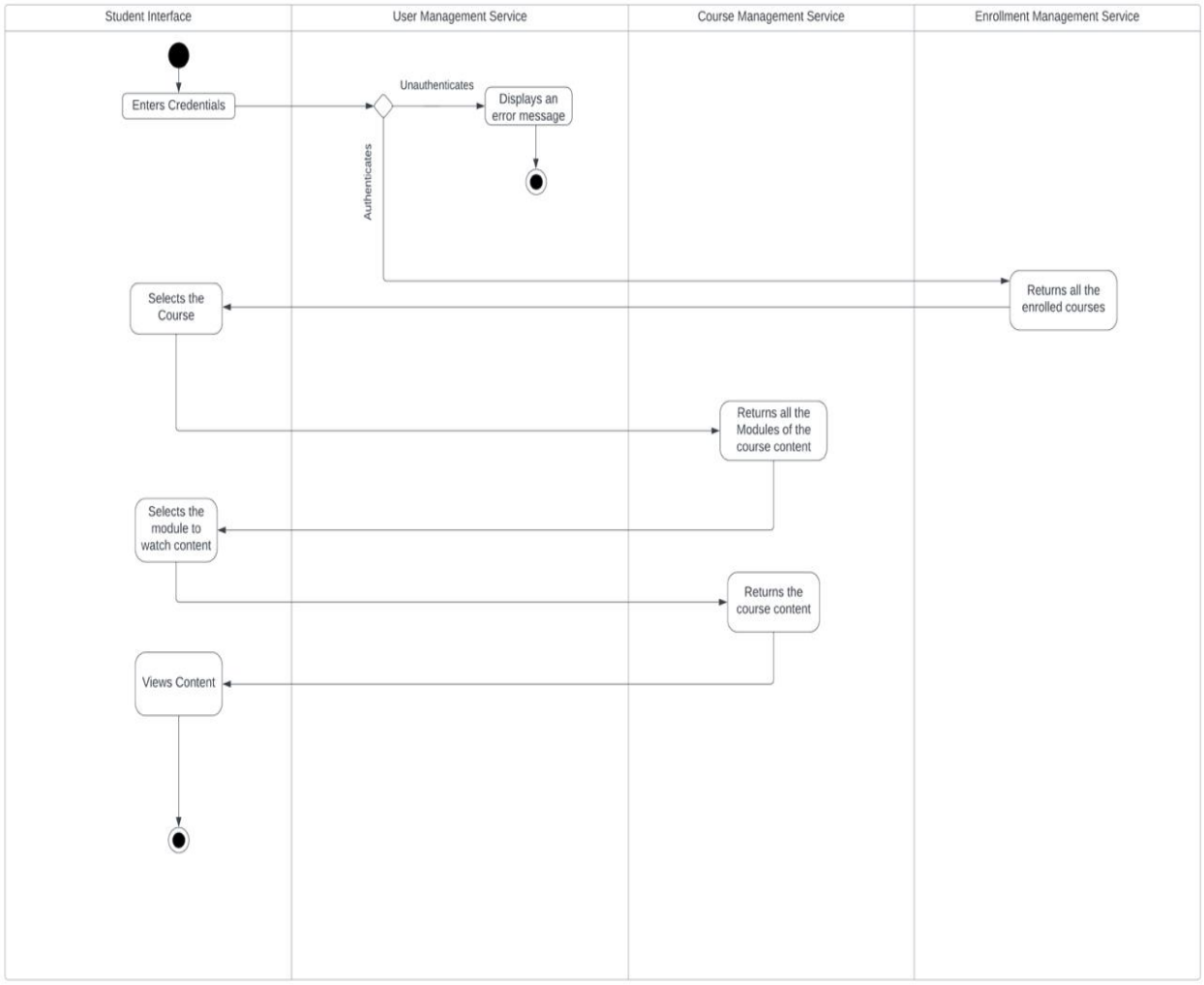
- Replication Strategy for Availability and Performance:
    - To address availability and performance requirements, we use replication for each microservice. Each microservice has a single replica, and load balancers are employed to distribute incoming requests. This strategy aligns with the microservices architectural style and the use of load-balancing mechanisms. It enhances availability by providing redundancy in case of node failures and optimizes performance by load distribution.
- Security (Encryption in User Management Service):
    - The User Management Service focuses on securely storing users' personal information. To achieve this, the service encrypts personal details before storing them in the database. This security mechanism ensures the highest level of data protection and aligns with security best practices.

ARCHITECTURE BEHAVIOR VIEW

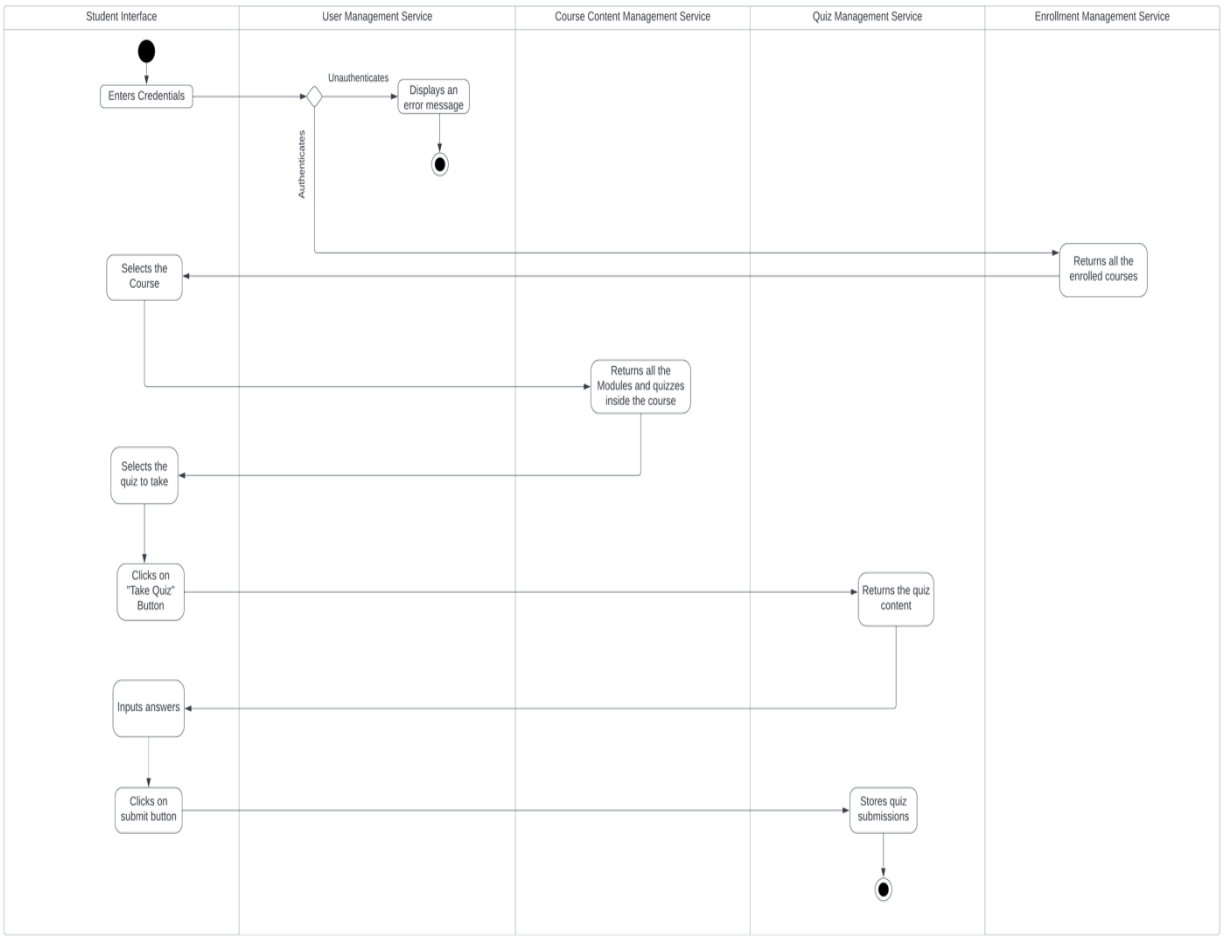
Use-case: Enroll in Course



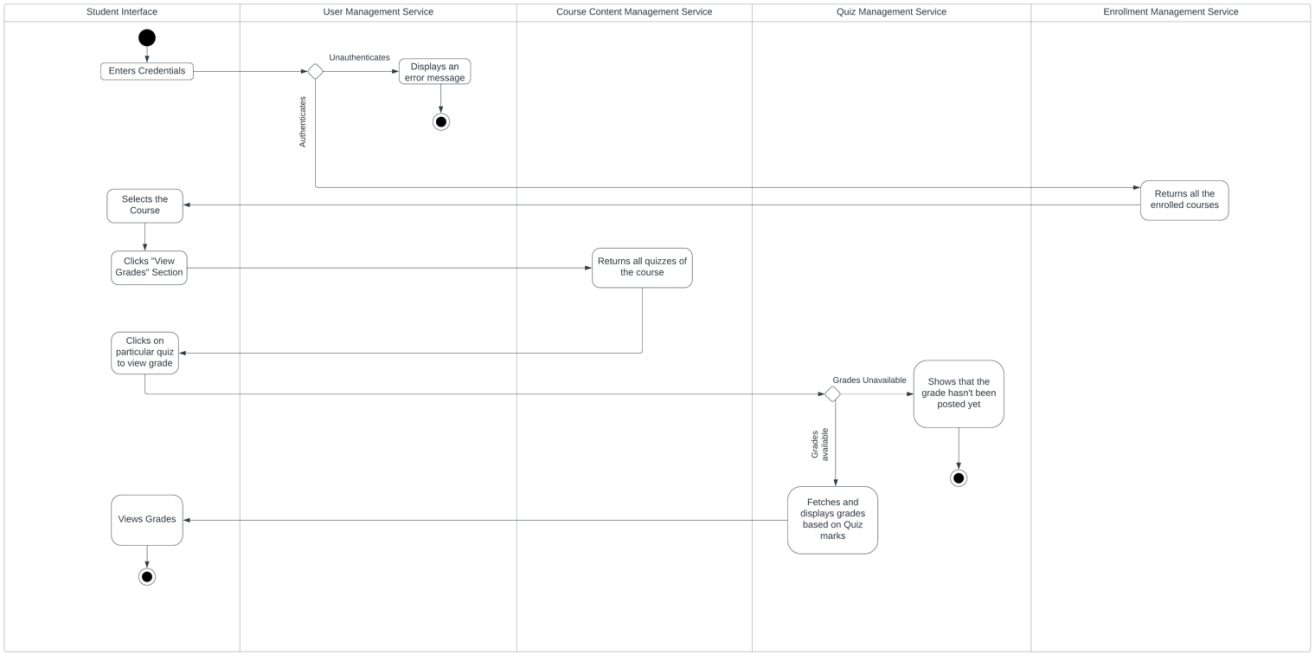
Use-case: View Course Content



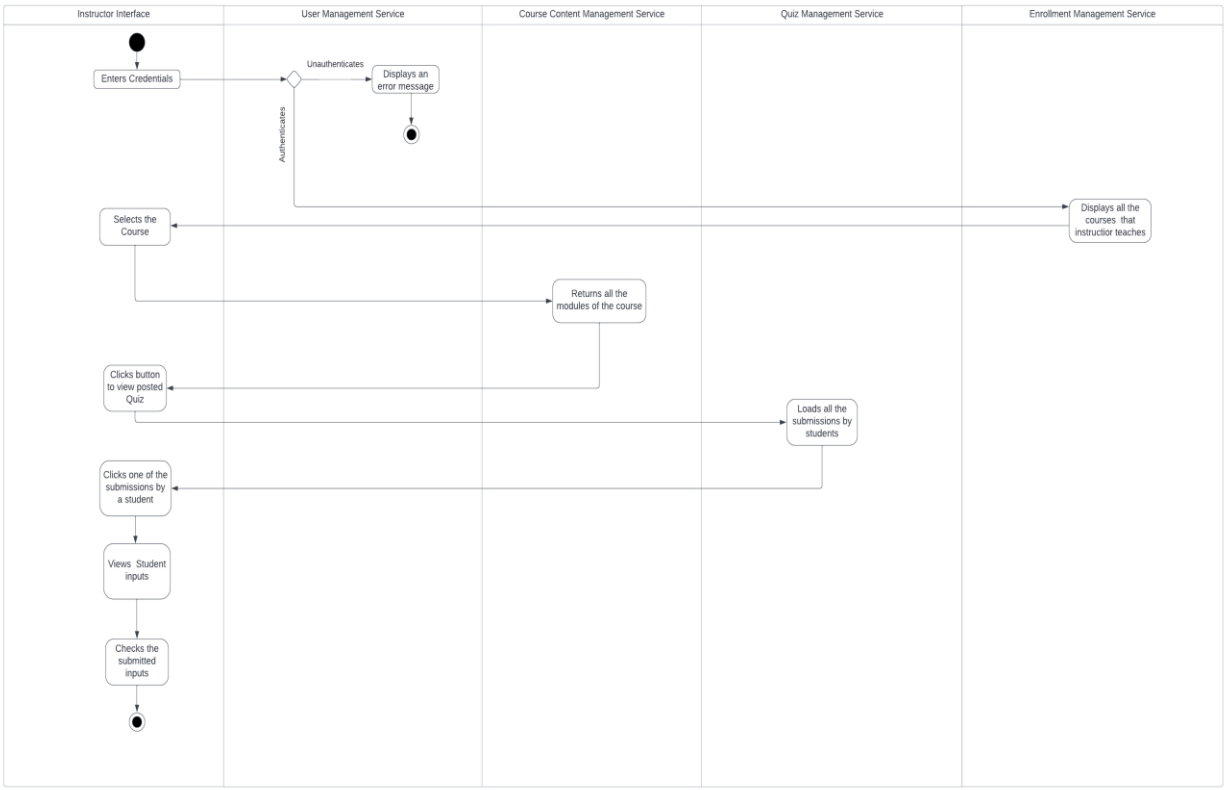
Use-case: Take Quiz



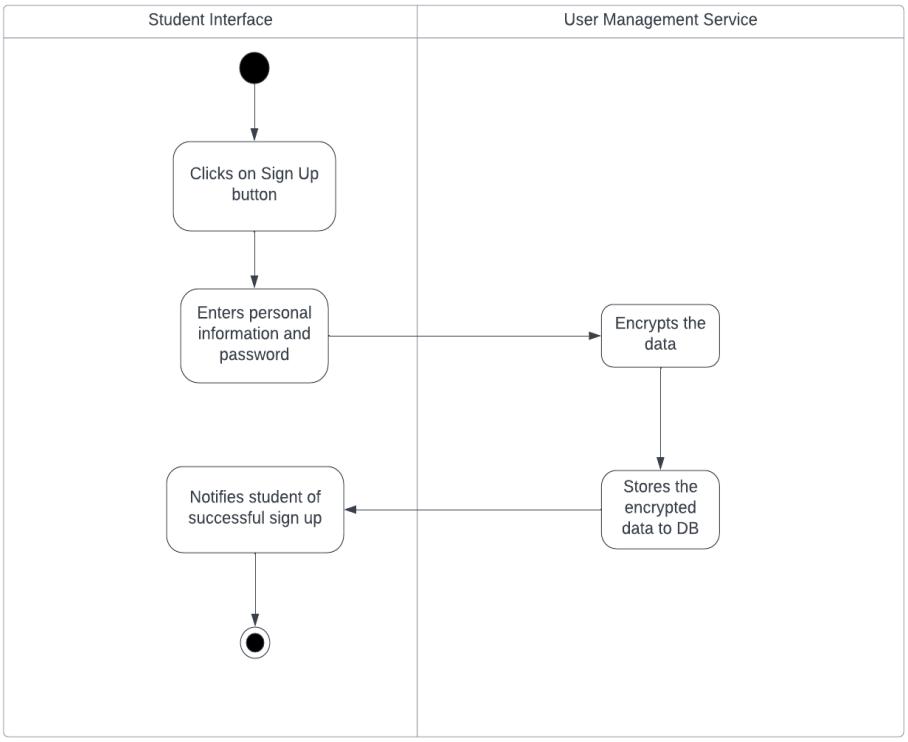
Use-case: View Grade



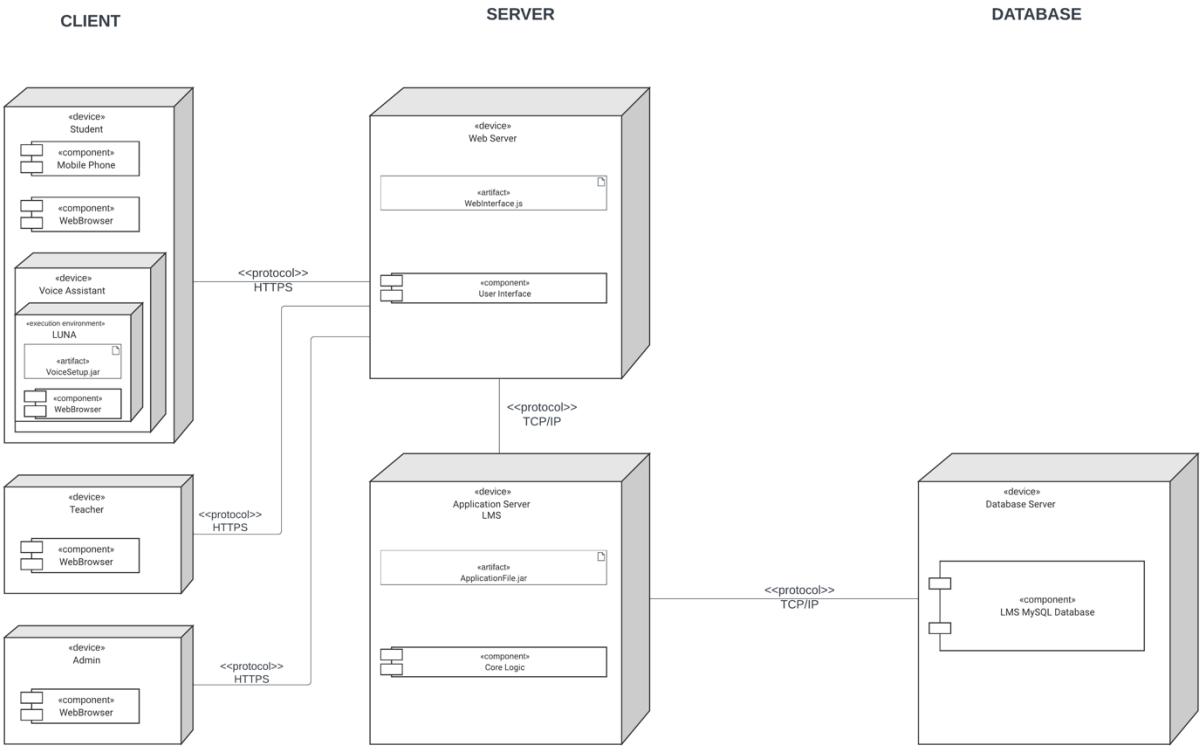
Use-case: View and Check Quiz (Instructor)



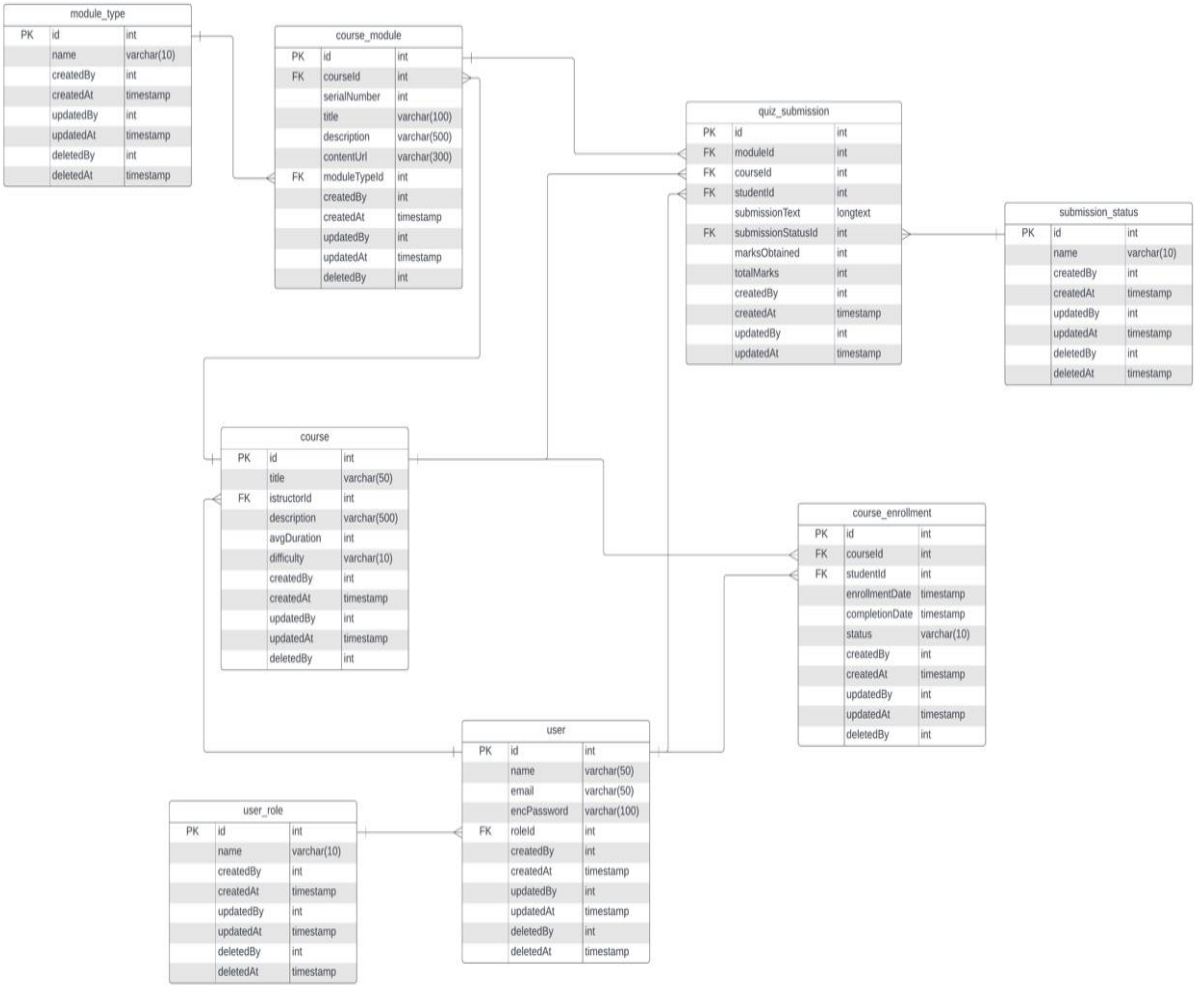
Use-case: Security (Quality scenario):



ARCHITECTURE DEPLOYMENT VIEW



INFORMATION VIEW



REQUIREMENTS ALLOCATION

	User Management Service	Course Content Management Service	Quiz Management Service	Enrollment Management Service	User Interface
Enroll in Course	X			X	X
View Course Content	X	X		X	X
Take Quiz	X	X	X	X	X
View Grades	X	X	X	X	X
View and Check Quiz	X	X	X	X	X
Security	X				X

ARCHITECTURE WORK ALLOCATION

	Jayesh Pamnani	Anjaneya Ketkar	Janesh Hasija	Divya Lnu	Manav Gupta
User Interface		Testing		Development	
User Management Service		Development	Testing		
Enrollment Management Service	Testing				Development
Quiz Management Service	Development				Testing
Course Content Management Service			Development	Testing	

DETAILED DESIGN KEY DECISIONS AND RATIONALE

One of the SOLID principles of object-oriented design we have applied in LanguageLift is the **Single Responsibility Principle** (SRP), which states that a class should only have one reason to change. The goal of SRP is to make code more modular with fewer interdependencies. By doing so, the code becomes more reusable and modular. The modularity in the software is introduced by dividing it into various components like:

- User Management Service
- Enrollment Management Service
- Course Content Management Service
- Quiz Management Service

Each of the following components/services performs a singular task, independently. Examining the constituent parts within LanguageLift according to the principle of single responsibility:

1. Modifications to user authentication or authorization may have an impact on the operation of quiz management if user management and quiz management were combined. Changes made to user-related functionality won't affect unrelated components in the quiz management system if they follow the Single Responsibility Principle.

2. Removing quiz administration from other elements of the system guarantees that changes to quiz scoring or updating quiz submissions won't affect other areas of the system. Every duty is included within a distinct service.

3. Changes made to the way users are added to or withdrawn from courses may unintentionally affect user-related features like user authentication, or authorization if enrollment management and user management are combined. For instance, Changes made to enrollment methods may unintentionally affect how user profiles are maintained if enrollment logic is combined with user profile management.

For LanguageLift, the **Chain of Responsibility** pattern makes sense since it facilitates the creation of an expandable and adaptable system in which different components may control different facets of the system's overall operation.

1. **User Management Service**: Using Chain of Responsibility, one handler might be designated to oversee a particular facet of user management while having several handlers. For instance, several handlers may handle different tasks, such as registration, profile updates, and authentication.

2. **Quiz Management Service**: We may use the CoR pattern to handle requests about the upload, modification, and scoring of quizzes. Depending on the chain, handlers may oversee various quiz kinds or phases of quiz administration (uploading, updating, scoring).

3. **Course Content Management Service**: In this case, requests for course creation, modification, and archiving might be handled by the CoR pattern. Each handler (Instructor, in this case) in the chain might have a specialty related to a certain facet of course administration, making the system expandable and adaptable.

Utilizing the Chain of Responsibility pattern has the following advantages:

1. **Flexibility**: Handlers are simple to add or delete without requiring changes to the client code.
2. **Decoupling**: The sender only moves the request down the chain; it is not tied to any classes.
3. **Reusability**: Different combinations of handlers can be used again for different processes.

In the context of LanguageLift, where various management services such as User Management, Quiz Management, and Course Content Management are integral components, the implementation of the **Facade Pattern** emerges as a strategic and beneficial choice. Here's why the Facade Pattern is well-suited for this software:

1. **Simplified Client Interaction**: The Facade Pattern provides a simplified and unified interface for clients, shielding them from the complexities of individual subsystems. In LanguageLift, users, instructors, or administrators interacting with the system can utilize specific facades without needing an in-depth understanding of the intricate processes within User Management, Quiz Management, or Course Content Management.

2. **Loose Coupling**: By employing the Facade Pattern, LanguageLift achieves a loose coupling between the client code and the subsystems. This loose coupling ensures that changes to the internal structure of subsystems do not necessitate modifications to the client code.

3. **Encapsulation and Abstraction**: Facades encapsulate the details of individual management services and present a high-level abstraction to clients. This encapsulation not only simplifies the client's perspective but also allows for modifications and enhancements within subsystems without affecting the external interfaces. It promotes a modular and scalable design.

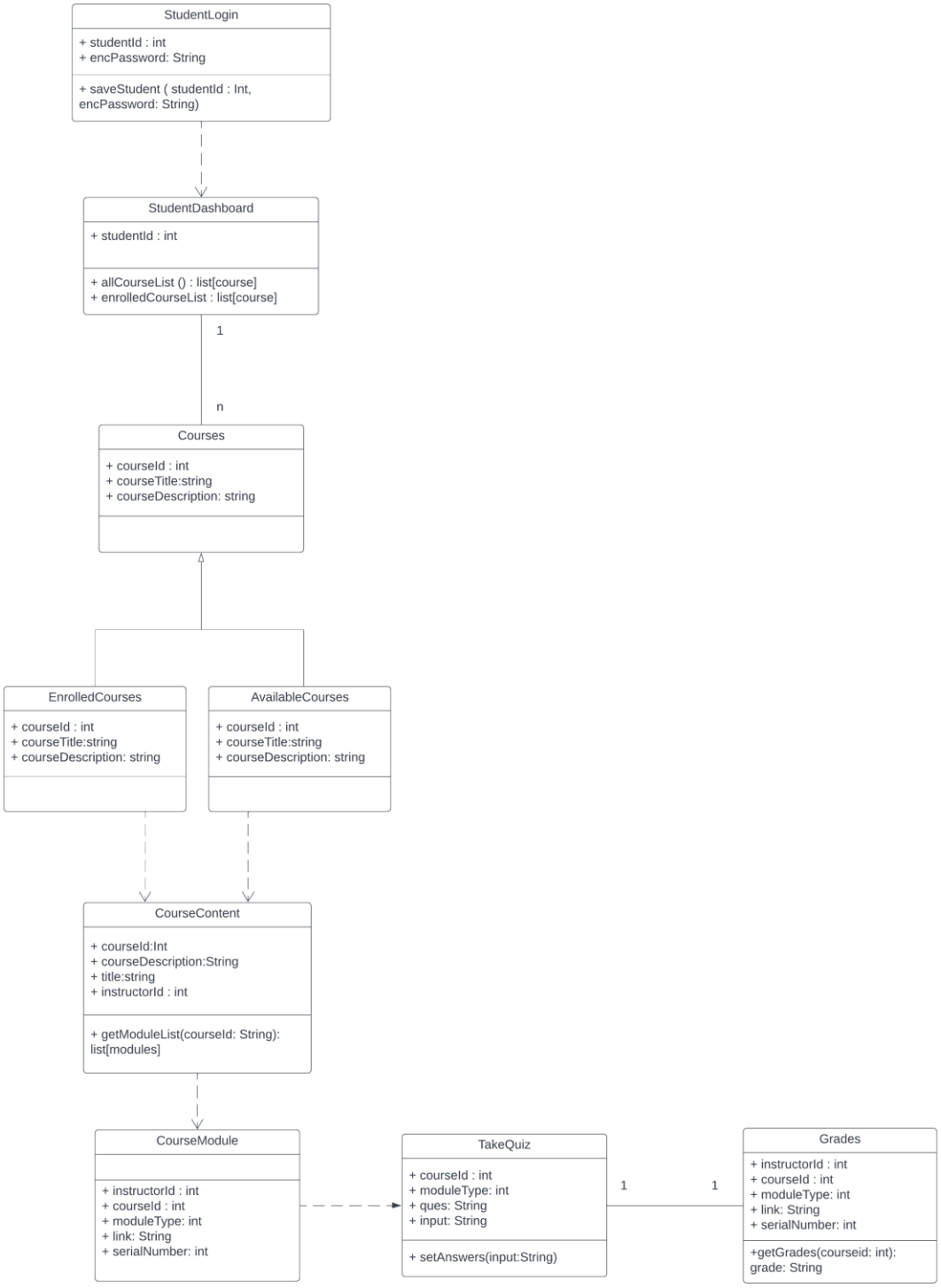
4. **Enhancing Readability and Maintainability**: By introducing facades for User Management, Quiz Management, and Course Content Management, the overall readability of the code improves. Client code becomes more focused and concise, leading to enhanced maintainability. This is particularly valuable in a language learning platform where the emphasis should be on educational content rather than intricate system interactions.

5. **Clear Separation of Concerns**: The Facade Pattern promotes a clear separation of concerns by isolating the complexities of subsystems. Each facade represents a distinct concern, whether it's user-related operations, quiz handling, or course content management. This separation simplifies development, testing, and maintenance activities.

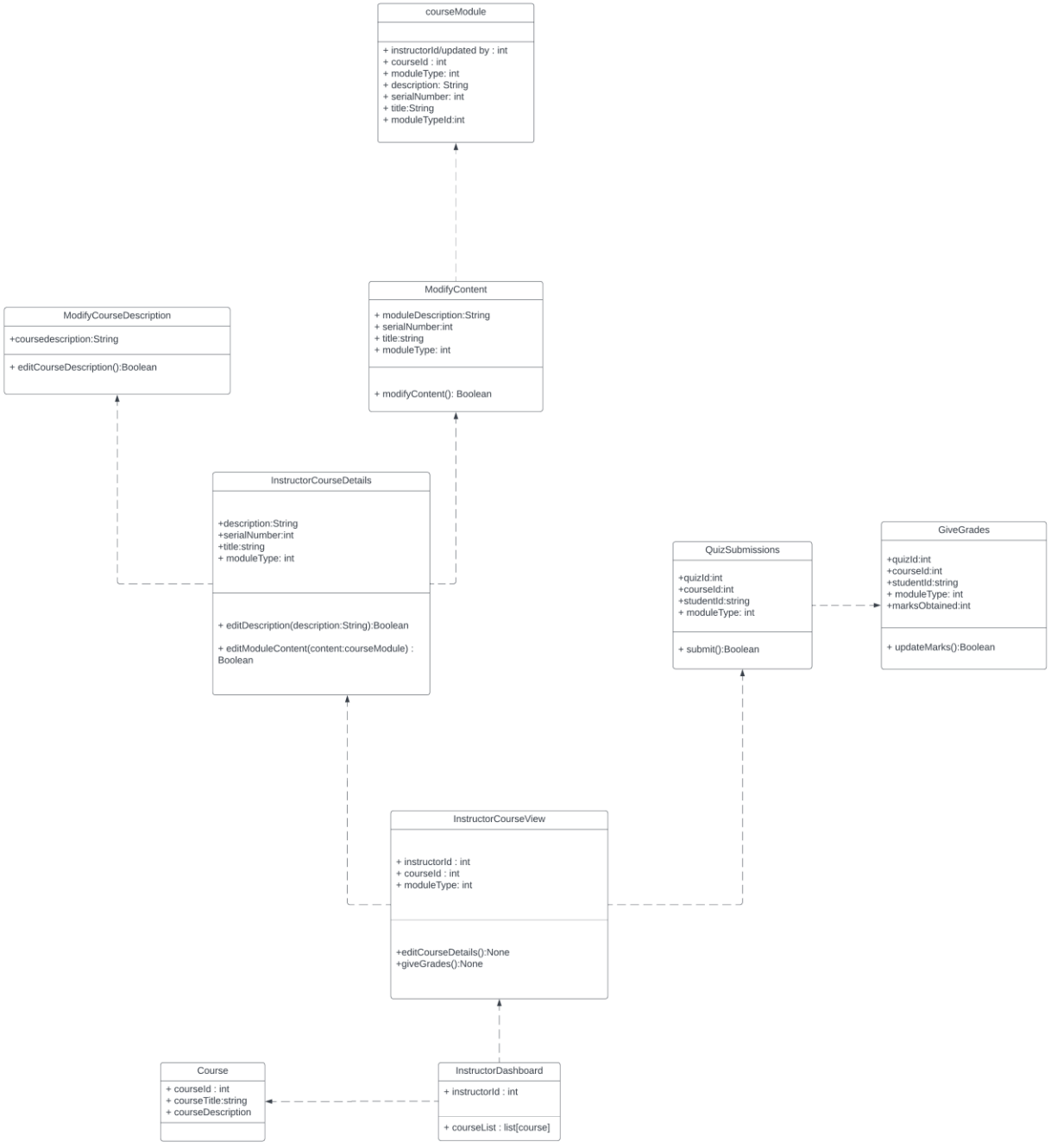
In conclusion, the Facade Pattern serves as a valuable architectural choice for LanguageLift, fostering simplicity, adaptability, and maintainability. It provides a structured and comprehensible way for clients to interact with the diverse management services offered by the language learning platform.

DETAILED DESIGN STRUCTURE

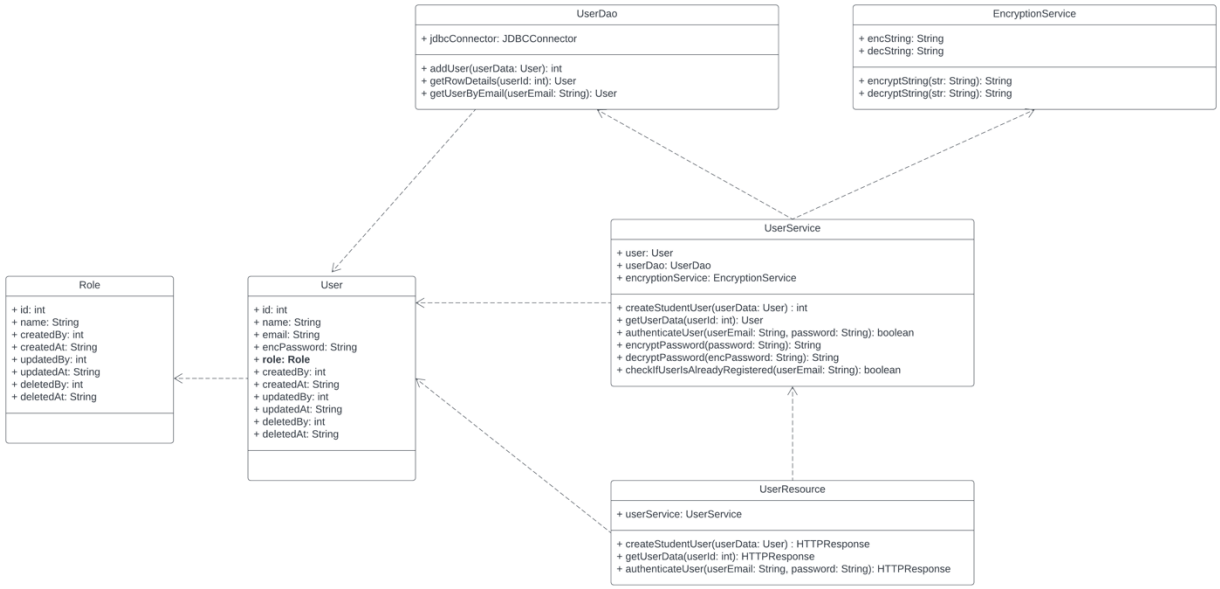
Student UI



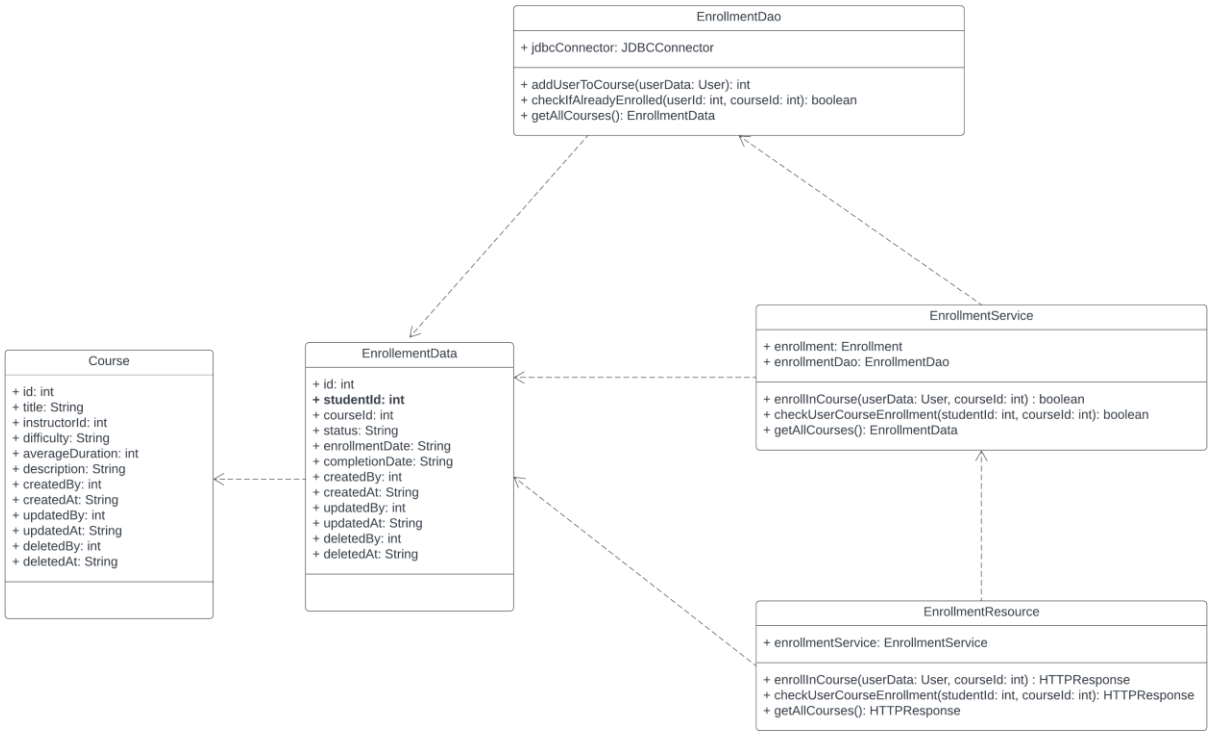
Instructor UI:



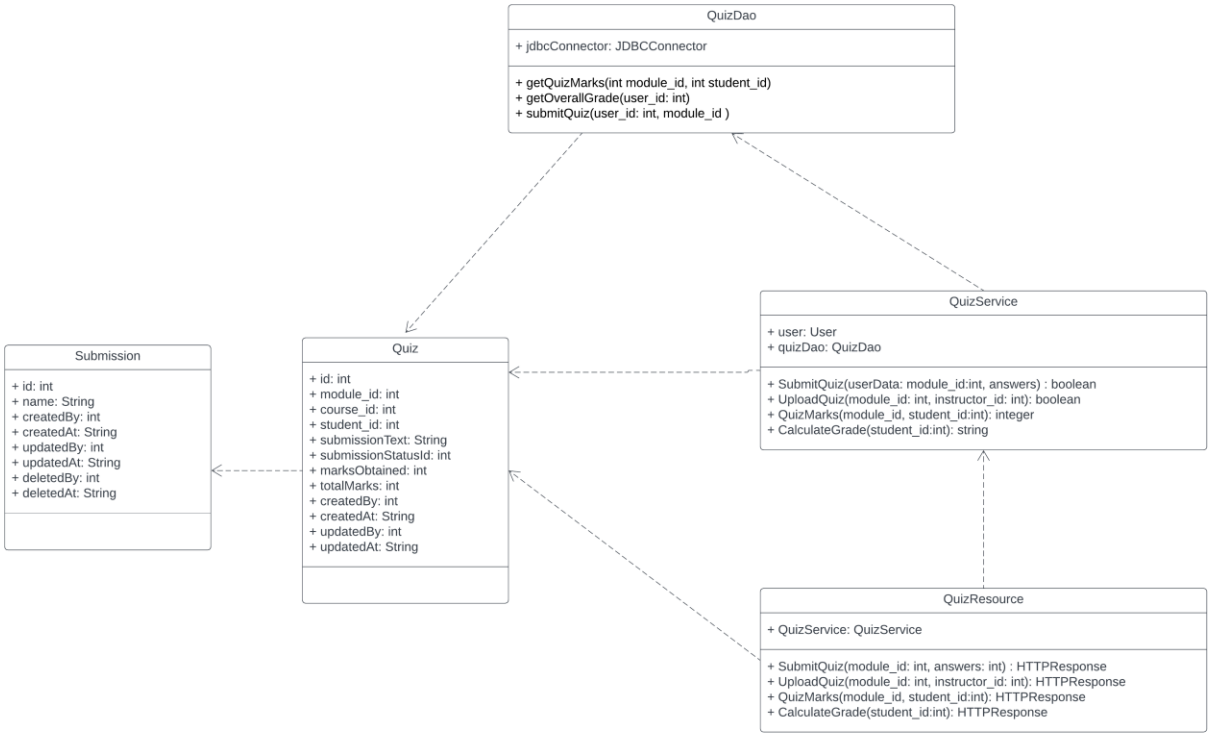
User Management Service:



Enrollment Management Service:



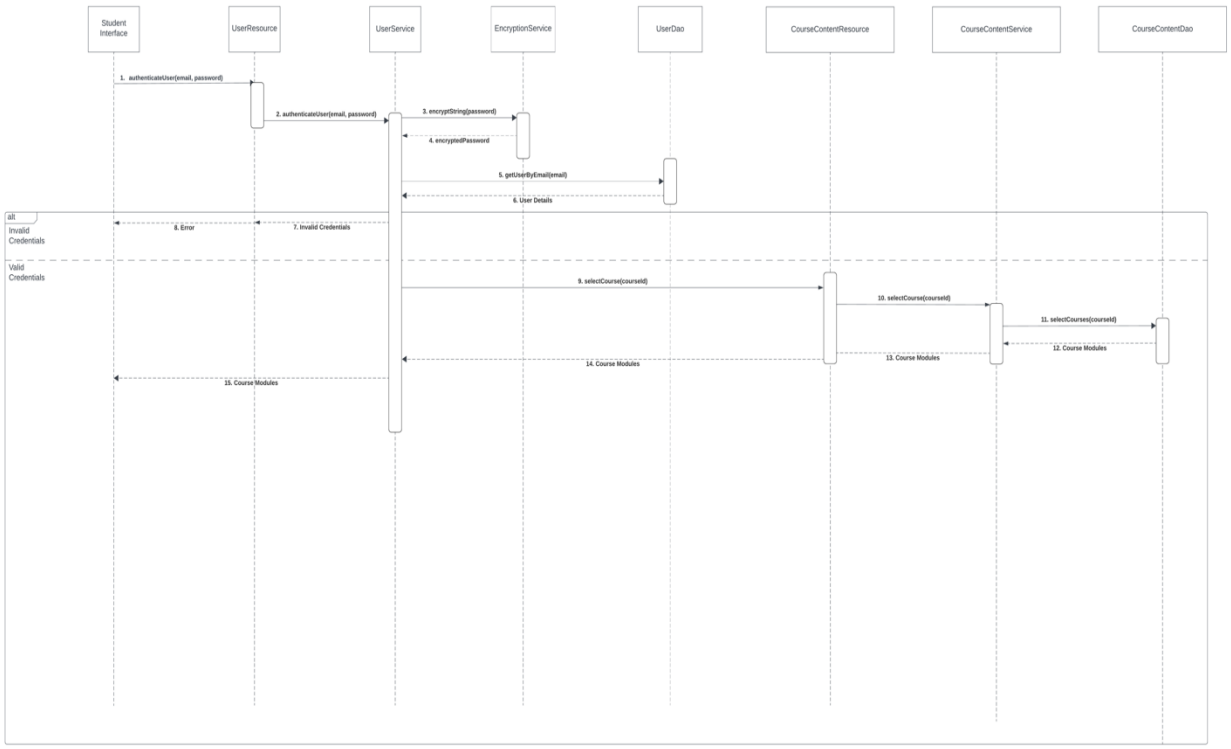
Quiz Management System:



Course Content Management Service:

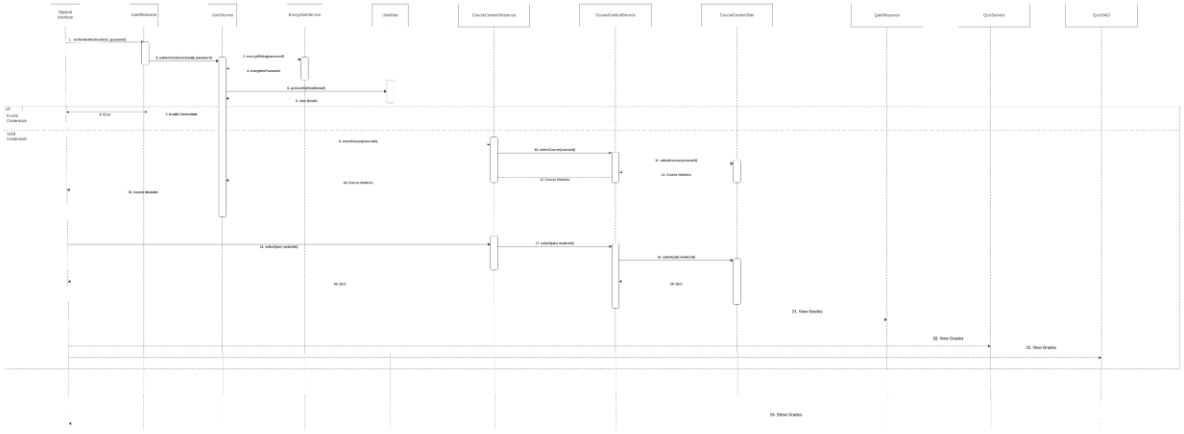
DETAILED DESIGN BEHAVIOR

View Course Content:



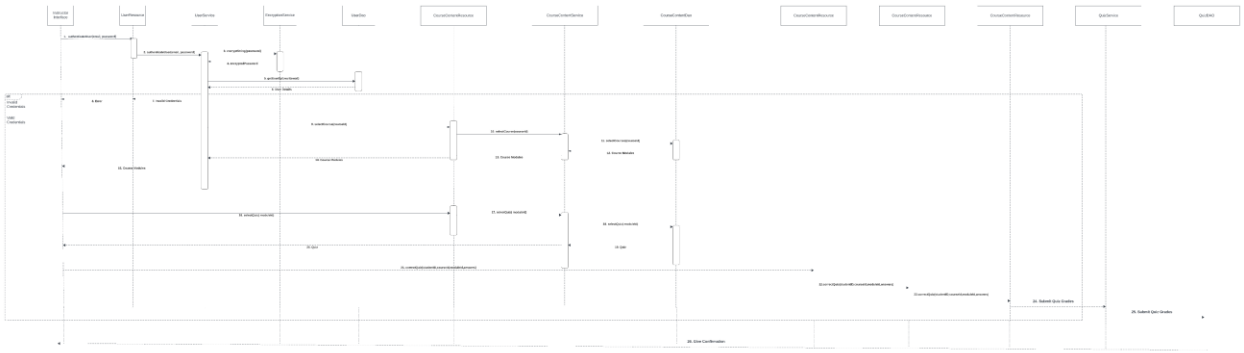
https://lucid.app/lucidchart/13681358-329e-4f78-975f-21762da71ed3/edit?viewport_loc=-2995%2C-776%2C5184%2C2434%2C0_0&invitationId=inv_24213632-e69b-480c-8c55-7997dd5c4361

View Grades:



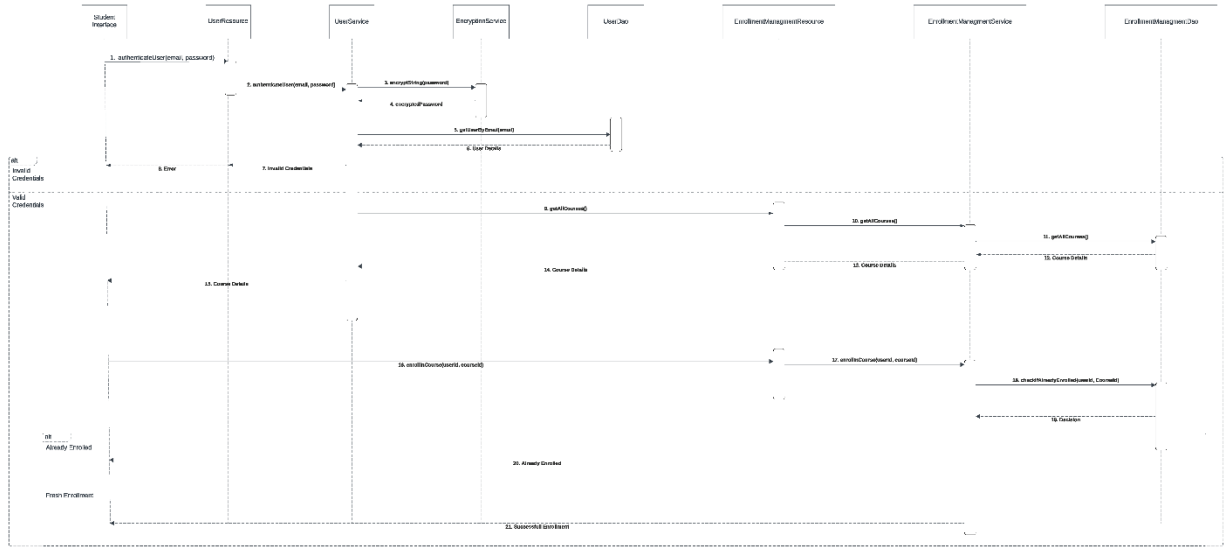
https://lucid.app/lucidchart/2d9fbefb-0982-4664-a95b-b3166e606d28/edit?viewport_loc=-55%2C-2%2C4573%2C2147%2C0_0&invitationId=inv_56151591-f6af-4118-a197-1f2e91313fa1

View and Check Quiz:



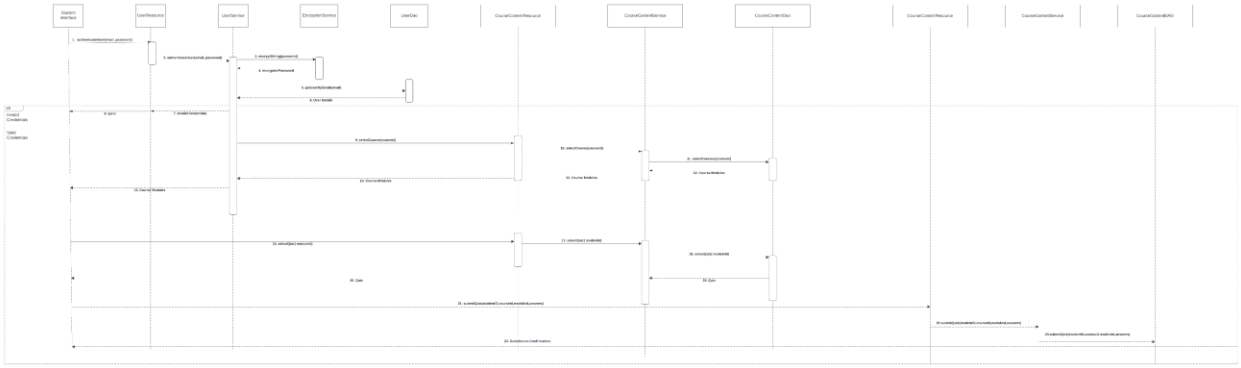
https://lucid.app/lucidchart/e1791f8e-d7c0-4299-aa5d-035d52a8c57f/edit?viewport_loc=1574%2C668%2C3502%2C1644%2C0_0&invitationId=inv_b2ff5b69-ceb8-4307-be42-7c45da4a540e

Enroll in Course:



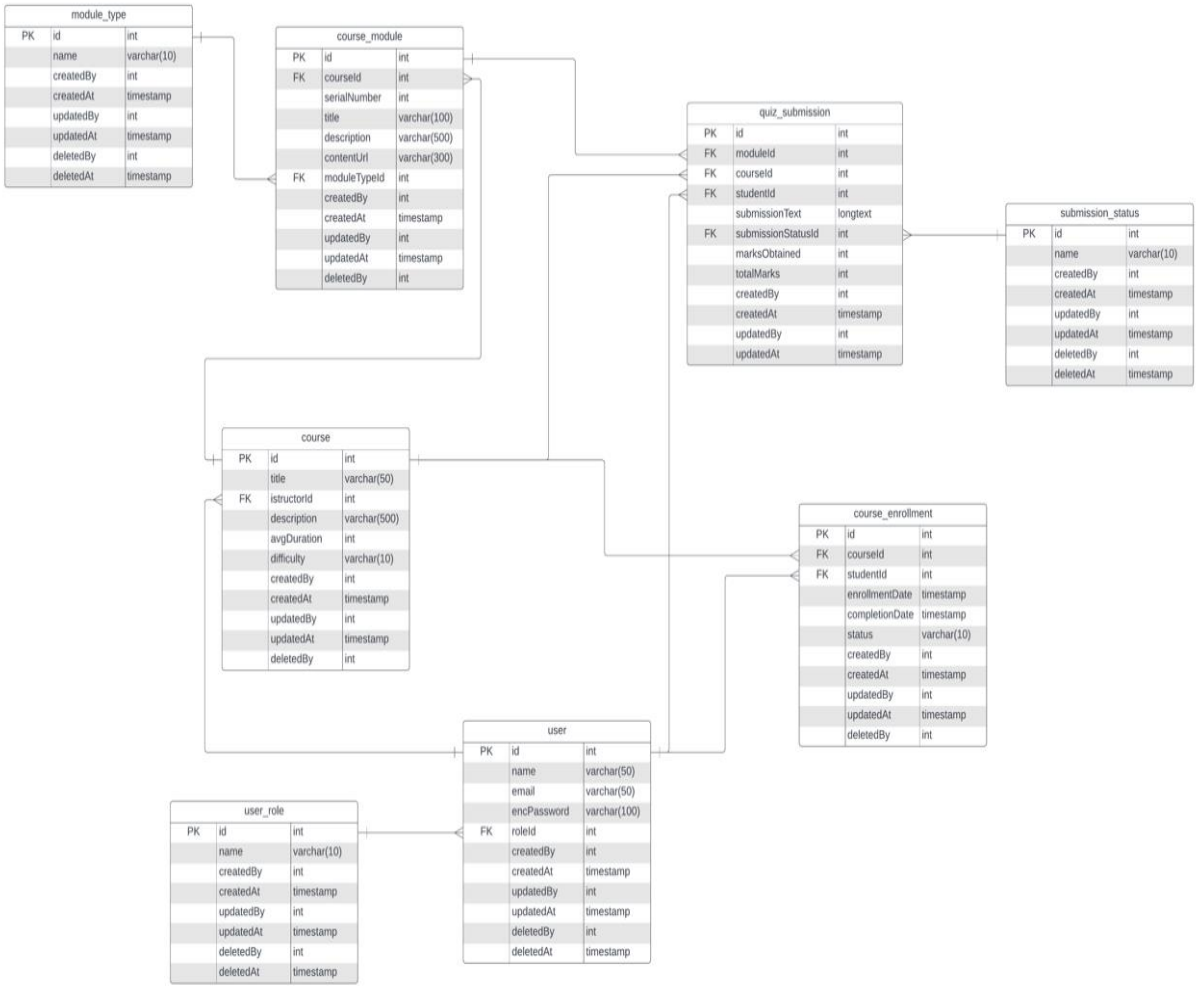
https://lucid.app/lucidchart/d9c3f22c-562a-4e1c-8598-98d2f38c8619/edit?viewport_loc=-3675%2C-1470%2C5534%2C3176%2C0_0&invitationId=inv_b0db4682-9a31-4914-b895-2094e2d8c95d

Take Quiz:



https://lucid.app/lucidchart/62b359a1-cad1-4f0c-873c-b5a97273249f/edit?viewport_loc=1024%2C-1019%2C3155%2C1481%2C0_0&invitationId=inv_35e36732-0144-4d3e-bc8e-5a4ae19b65b9

PHYSICAL DATA MODEL



ER Diagram Description
The ER is an Information View of the database for LanguageLift.

The entities are:

- **user** - This entity stores information about the system's users such as their names and email addresses.
- **user_role** – This entity stores information about the different roles a user can be assigned, such as student and instructor.
- **course** - This entity stores information about the courses that are offered, such as the course name and description.
- **course_enrollment** – This entity stores the enrollment details of students for various courses.
- **course_module** – This entity stores information about the different modules the user needs to complete to complete the course.
- **module_type** - This entity stores the different types of modules that can be used in LMS courses, such as quizzes, and study material.
- **quiz_submission** - This entity stores the student's submissions to the quizzes, including their answers to each question and their score.
- **submission_status** – This entity stores various statuses that a quiz submission can have such as Not Graded and Graded.

The relationships between the entities are as follows:

- **user and user_role** – A role can be associated with many users but, a user can have only one role.
- **user and course_enrollment** – A student can be enrolled in many courses, but one-course enrollment would be associated with only one student.
- **user and course** – An instructor can teach multiple courses, but a course can be taught by only one instructor.
- **user and quiz_submission** – A student can have many quiz submissions of different modules, but a quiz submission would be associated with only one student.
- **course and course_enrollment** – A course can have many enrollments, but an enrollment must be associated with only one course.
- **course and quiz_submission** – A course can have multiple quiz submissions, but a quiz submission can only be associated with one course.
- **quiz_submission and submission_status** – A quiz submission must have only one status, but status can be acquired by multiple quiz submissions.
- **course_module and quiz_submission** - A module can have multiple quiz submissions, but a quiz submission can only be associated with one module.
- Page 15 of 17

- **course and course_module** – A course can have multiple modules, but a module would be associated with only one course.
- **module_type and course_module** – A module can have only one type, but a type of module can be associated with many course modules.

The primary keys for each entity are:

- user - id
- user_role – id
- course - id
- course_enrollment – id
- course_module – id
- module_type - id
- quiz_submission - id
- submission_status – id

The Foreign keys for each entity are:

- **user** – roleId (references the id attribute of user_role entity)
- **course** - instructorId (references the id attribute of the user entity)
- **course_enrollment** – courseId (references the id attribute of course entity), studentId (references the id attribute of user entity)
- **course_module** – courseId (references the id attribute of course entity), moduleTypeId (references the id attribute of module_type entity)
- **quiz_submission** - courseId (references the id attribute of course entity), studentId (references the id attribute of user entity), moduleId (references the id attribute of course_module entity), submissionStatusId (references the id attribute of submission_status entity)

This ER diagram can be used to create a relational database for the LMS system. Each entity would be represented by a table in the database, and the relationships between the entities would be represented by foreign keys.

ALGORITHMS

Algorithm Used for Encryption - AES

Implementing AES encryption for storing user credentials in LanguageLift holds several advantages specific to the nature of the language learning platform:

1. User Trust and Confidentiality:
   LanguageLift deals with sensitive user information, including progress data and personal details. AES encryption ensures that user credentials are stored securely, fostering trust among language learners who rely on the platform for their language education.

2. Multilingual User Base:
   As a language learning platform, LanguageLift caters to a diverse and potentially global audience. AES encryption's widespread acceptance and compatibility across different languages and platforms ensure seamless integration without compromising security.

3. Alignment with Industry Best Practices:
   AES encryption is considered an industry best practice for securing sensitive information. Its implementation in LanguageLift aligns with established security standards, reflecting a commitment to providing users with a secure and reliable language learning environment.

4. Integration with Personalized Learning Features:
   LanguageLift's emphasis on personalized learning experiences requires secure handling of individual user data. AES encryption enables the secure storage and retrieval of user-specific information, contributing to the platform's ability to deliver tailored language courses.

5. Long-term Data Integrity:
   As LanguageLift accumulates user data over time, ensuring the long-term integrity of this information is crucial. AES, with its proven long-term viability, provides LanguageLift with a robust encryption solution that evolves with the platform's growth.

Implementing AES encryption specifically for user credentials in LanguageLift enhances the platform's security, aligns with industry-specific considerations, and contributes to a trustworthy and personalized language learning experience for users.

ARCHITECTURE TO DETAILED DESIGN TRACING

	User Resource	User Service	User DAO	User	Role	Enrollment Resource	Enrollment Service	Enrollment DAO	Course
User Management Service	X	X	X	X	X				
Enrollment Management Service						X	X	X	X
Course Content Management Service									
Quiz Management Service									
User Interface									

	Quiz Resource	Quiz Service	Quiz DAO	Quiz	Submission	Course Content Resource	Course Content Service
User Management Service							
Enrollment Management Service							
Course Content Management Service						X	X
Quiz Management Service	X	X	X	X	X		
User Interface							

	Course Content DAO	Module	Module Type	Student User Interface	Instructor User Interface
User Management Service					
Enrollment Management Service					
Course Content Management Service	X	X	X		
Quiz Management Service					
User Interface				X	X